

# Interactive graphic consoles—Environment and software

by ROBERT L. BECKERMEYER

*Research Laboratories, General Motors Corporation  
Warren, Michigan*

## INTRODUCTION

The usual software support for graphic consoles does not provide system services designed for the console user who is a production-oriented application expert. This paper describes a console environment and high-level language with a supporting operating system designed for the application expert.

The aim has been to improve four troublesome areas: programming a new application, defining input data for functional routines, screen management and dynamic error recovery.

The importance and acute need for improved software of this type is substantiated by the large number of such systems being designed and languages proposed. RAND Corporation is developing a programmer-oriented graphics operation.<sup>1</sup> At the University of Utah, W. M. Newman is working on a high-level programming system for remote graphic terminals,<sup>2</sup> and A. van Dam at Brown University is working on a hypertext editing system.<sup>3</sup> The list is very long.

At the G.M. Research Laboratories, experience with the DAC system<sup>4</sup> has enabled us to evolve a console environment incorporating those human factors in man-machine communication which we have found to be best. This paper describes the principal parts of that environment with emphasis on external appearance.

The material is divided into four main sections: graphic console environment, programming language, procedure execution, and operating system. The principal features of the DAC programming services and console man-machine communication techniques will be outlined for background information. The system facility and console appearance for selection of values for input data will be described and illustrated. Control of displays and interaction with procedure\* execution will also be discussed. Procedures use console displays for a two-way communication instrument between user and machine.

\* The term procedure in this paper is a collection of computer programs or subroutines linked with a logic structure designed for several variations of a problem solution.

This environment is based on the belief that people solving a problem using a graphic console should determine as much of the display and execution logic as possible. By making the programming language for display generation and procedure logic use exactly the communication techniques used during problem-solving sessions, the application expert feels more at ease in a programming situation. An additional strong aid to the programming task is the automatic retrospective programming provided by the operating system. This allows a console user to "capture" the solution procedure as it evolves during his problem-solving session.

Program execution is done interpretively on an internal structure that is, by design, amenable to change. Most graphic console applications are "cut and try" and highly dependent on the console user's background and experience. The approach and problem solution has to be individualistic. With a flexible internal structure the solutions can be altered many times so the final procedure is tailored by the console user.

An operating system to manage screen displays, accept all user interrupts and supervise execution will be described in terms of the fundamental modules within it. This system allows data to be defined by a new technique called "association." Association enforces a uniformity in communication that helps to insure usability of new programs without much experience by the user. And yet the system is in no way dependent on the actual appearance of the displays or assignments of function buttons.\* The operating system\*\* is table driven, allowing it to service different applications and many consoles simultaneously without restricting the

\* A function button is a key or button that may be programmed to link to a particular computer code when pushed by the console user. These buttons may be special keys on the alphanumeric keyboard or collected together in a separate box on the console.

\*\* Operating system in this paper refers not to the basic system support like OS/360, but to a collection of routines designed to interpret procedure internal structure, handle all data associated with this structure and interpret all the console user interactions with functional parts of the console (keyboard, function button or light pen).

flexibility required by the user and at the same time keeping the operations in the different consoles separated.

Included is a complete example illustrating the principal parts of this environment. Although only a small portion of the language symbols are used in the example, those used are among the most important.

## GRAPHIC CONSOLE ENVIRONMENT

### *Long-range objective*

A seven-year study<sup>4,5</sup> on computer graphics with emphasis on programming efficiency has led to the long-range objective of a more direct path for console users to design solutions for their problems. Many applications have been considered including body design, statistical analysis, information retrieval and project planning and control. A characteristic of these applications is that the procedures continuously change to incorporate improvements and new approaches. For programming efficiency, a high-level language allowing the application expert to program in an environment very close to that in which they solve application problems is needed.

### *DAC background*

For DAC<sup>4</sup> system programmers designed what is called the Descriptive Geometry Language (DGL). The DGL statement

LN7, PT8, PT9, UF2 = INLIN(SU3, Y = 20.), RG2

makes symbolic references to geometric data. It also uses a package of functional evaluations and subroutines to do transformations, generate new data and other geometric operations on the data. In DAC these subroutines are called operators. In the DGL statement above, the INLIN operator is used to find intersections of surfaces and lines. We take the surface SU3 and the plane Y = 20 and form an intersection over the range RG2. The intersection line is LN7 with end points PT8 and PT9 and the surface normal unit vector function UF2. A large number of such operators are available to designers. All were written by G.M. Research mathematicians.

Designers and engineers use these operators in statements to write a computer program or procedure. The system then interprets these statements one by one

in an interpretive mode. The designer or engineer, when at the console, can start and stop at any statement in his procedure. If need be he can back up and reexecute any number of the statements to correct or modify the design. Graphic Displays on the DAC console screen provide for review and evaluation of the results. We found, however, that application specialists were not amenable to the discipline of a written programming language and a search for a method of defining procedures in a manner more natural to the users continued.

In the next system, groups of operators were made available simultaneously via DAC console function buttons (these are similar to those on the IBM 2250 function keyboard). The groups were called "overlays"<sup>5</sup> because as each group became available, a celluloid sheet with identifying labels for that group was placed over the keyboard. In general, operators were grouped by function; for example, the SURFACE overlay operators dealt with surface definition problems. There were eight overlays in all. The coding for these operators was done by system programmers and required 13 to 30 man-months per overlay. Using a combination of Descriptive Geometry Language procedures and overlays, the DAC system is now being used heavily for production work.

Based on our experience, the next development was to communicate with the user through pictogram<sup>6</sup> representations of the operators. Through considerable experimentation with DAC-experienced users we were led to the amalgam of proven DAC techniques and pictogram communication which constitutes our present console environment.

### *Improved console environment*

The new environment provides the same operators as were available through the function buttons of an overlay in the previous system. The improvement is obtained by pictorial communication between the mathematical function routines stored internally to the computer and the man sitting at the console who wants to use them in a problem solution.

Screen management is carried out by dividing the console display area into three function areas. A typical screen format is shown in Figure 1. The upper area is called the work area. The lower right-hand area is known as the operator area and in some ways corresponds to the celluloid overlay of DAC. The lower left-hand area is known as the control area. The operating system distinguishes between console user actions in these areas and responds accordingly. The display in the work area shows three parts of a design problem: a windshield surface, some lines for the windshield wiper pattern and

the rear view mirror subassembly.\* As the design is evolved, the modifications and additions are shown in this area of the screen.

The operator area in Figure 1 contains only text describing intersection operations and thus wastes communication potential of a graphic console. Contrast this with the operator area shown in Figure 2 where a pictorial representation, a pictogram (6, page 384), of the functional capabilities of the operator realizes more of this power.

The pictograms in the operator area show three types of geometric intersections to determining an intersection point. These illustrations, if well done, have a "universality" that will transcend the limitations of the prose equivalent shown in Figure 1. The differences between these types of intersections and the geometric elements involved is readily understood by those trained in elementary geometry regardless of the way in which they express the concept, as a mathematician or designer. Consider the problem of designing a tool used for automobile body panels called a die shoe and having people from different departments working on problems related to making this die shoe. Different jargon may be used by the departments, but all persons will recognize a blueprint or stylized drawing (or pictogram) of this part. A simplification of the drawing of this part with emphasis on the possible design operations can be shown

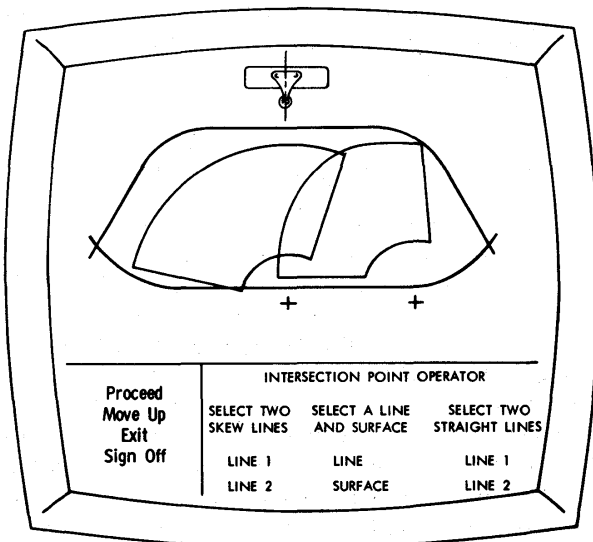


Figure 1—Display area format

\* A subassembly is the automobile body designer's term for a detail drawing of a subpart of a major assembly. For example, a door lock subassembly of the door panel.

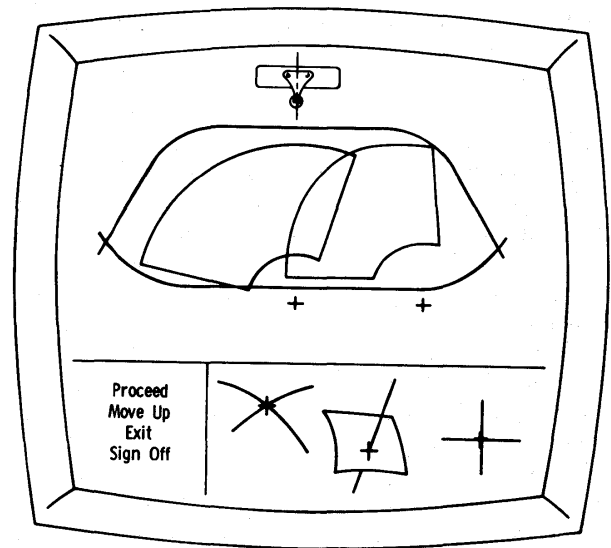


Figure 2—External appearance of a graphic operator

in the operator area and the operator will be a useful tool for people in any of the involved departments.

In the control area there are four possible controls. The console user transmits his wishes for display control and execution interruption to the operating system through this area. Use of the three functional areas is described in more detail below.

Suppose the user wished to find the intersection of a surface and a line. These two parameters are indicated in the center pictogram which represents this operation. The user must supply the values for the parameters from the work area display, however. The order in which the values are defined is arbitrary. The user starts by selecting with the light pen either the surface or the line in the middle pictogram. This first selection tells the system two things, which type of intersection is to be calculated and which parameter is to be supplied first.

The choice of parameters is done by associating each stylized parameter in the pictogram with an actual counterpart in the work area, using the light pen to indicate the two members of the correspondence. The first part of the association is made in the pictogram allowing the system to limit the user's choice in the work area to elements of compatible type. This prevents such erroneous associations as a line parameter with a surface element. This error prevention is done using the technique of selective enabling described below. As soon as all the parameters for the computation have been defined, the system calls the appropriate subroutine passing the work area line and surface chosen by the

user. If mathematically possible, the subroutine computes the intersection and passes the intersection point back to the operator for display in the work area of the screen.

At any time prior to the completion of parameter definition and the subroutine call, the user may redefine any of the data values by a simple reassociation. He first selects the parameter in the pictogram that is to be redefined and the system prepares to override the work area data choice made earlier. The user's second selection of a work area item completes the parameter redefinition.

In the windshield example, the user might associate the line parameter in the pictogram with the center line of the rear view mirror and the surface parameter with the windshield surface. The windshield surface is selected by placing the light pen over any part of the boundary lines of the windshield.

The light pen selections make use of both selective enabling and reactive displays.<sup>7</sup> On display in the work area of the example are lines, surfaces and points. The operator can cause the system to enable for selection lines only, points only, surfaces only or any combination. This selective enabling is controlled by information stored with each parameter in the operator and passed to the system when the user selects a parameter in a pictogram. All attempts to select items which are not enabled are ignored. Selective enabling is not used in the operator area display; all parameter representations in this area are always enabled, allowing the user to respecify parameters should he make an error.

The reactive display technique allows the user to ensure that the intended element is being selected. This is particularly important in crowded parts of the work area where elements are close together. When a light pen is over an enabled item, it is intensified and will remain so until it is selected by removal of the pen from the screen or until the pen is moved away from the item. The user may thus adjust the position of the pen on the screen until the correct element is intensified and then make his selection. As a further aid to the user, only enabled items are intensified.

The words on display in the control area represent functions which can be executed independently of the current operator and of the function being performed by the operator. For instance, if the user selects "MOVE UP," the operator is interrupted and the items on display in the work area are shifted or translated up some predetermined distance.

If an error occurs while using any operator, an appropriate message can be put on display in the operator area and the entire interpretation of the operator will stop until the user acknowledges the error with the light pen. Thus he has an opportunity to take

some immediate corrective action and continue. Response to an error might involve a change to a parameter value and reexecution of that part of the operator or the current operator may be pushed down and another operator called up to generate new data. This new data may then be used to replace some parameter value that caused the error condition. The operator may also be programmed to open an auxiliary path allowing the user to supply extra data needed to overcome the error condition. The system can provide some standard error escapes, but the operator can be programmed to have almost an unlimited variety of error recovery paths dynamically selectable by the user.

With the same operating system supporting all applications for screen management and interrupt processing, data selection by association becomes paramount and can be fine-tuned giving the console user good response to his requests. All operators have a similar internal structure: an APL<sup>8</sup> structure generated by APL statements within a system module from the graphic operator source program. This structure is illustrated for the example in Figure 2 later in the paper. With similar internal structures a more uniform approach to the console environment can be assured. Uniformity is important because any departure from a conventional course of action is a distraction for the user. This became evident while observing DAC users working in the overlay environment. Each overlay was considered an experiment in human factors so the standard course of action varied from overlay to overlay and resulted in confusion for the user when switching overlays.

## OPERATOR PROGRAMMING LANGUAGE

### *Operator programming*

The operator in the previous example has shown that an operator must be able to present displays in the operator area and associate each parameter of the operator with the data selected by the console user. An operator must also be able to invoke subroutines and other operators, display the output from a subroutine call, and give the console user the ability to select alternative ways to solve his problem. The flow of logic in an operator is such that a subroutine will not be called until all its parameters have been given values. As soon as the last parameter receives its value the subroutine is called. The user has freedom of choice of the order in which he specifies parameters, thus he may define values for the parameters of several subroutines in a mingled sequence. For this reason the order

in which the statements of an operator will be executed is indeterminate at the time of programming. This contrasts with the usual type of program where statements are executed in strict sequence except for branching. The language used to program operators must not only provide for simultaneous flow of data and logic, but must allow an operator to begin execution at almost any statement. Once the initial display for an operator is made in the operator area, the user's selection of the available parameters determines where the operating system starts operator interpretation.

### Graphical operator language and representation

The illustration of Figure 3 shows both the graphic operator interface (operator picture) as before and a graphic representation of the operator program, the programmer's view of the operator. This program representation is called a logic diagram since it describes the logic flow of the operator. It is the programmer's source program and is never shown to the console user in an application problem-solving situation; however, since operators are programmed at the screen in the same way in which the application user solves his problem, this diagram forms the work area display for the operator programmer. The programming language for operators is really a set of operators which add symbols to the logic diagram and build the structure to be used by the interpreter during execution. In Figure 3 dashed lines show the connection between symbols in the diagram that represent data parameters and their graphic display representation in the operator area.

This operator consists of three alternative ways of generating an intersection point. Each alternative is shown as a horizontal line of symbols and ends with an X symbol. Each line of symbols is referred to as an alternative path. The upper path, represented in the operator picture by the left illustration, covers the case of intersecting two skew lines, and is made up of two AS symbols, a CL and a DS symbol. The middle path represents an intersection of a line and surface and the lower path the intersection of two straight lines. The three alternative paths are joined by a vertical line to the AL or alternative symbol. With the AL symbol the console user is free to select the intersection construction that fits his needs for the problem at hand. Any one of the parameters may be selected to start the operator and this selection will determine which alternative is to be used.

In the first alternative path, the AS symbol representing a data parameter denotes a data association as described above. Part of the information stored with

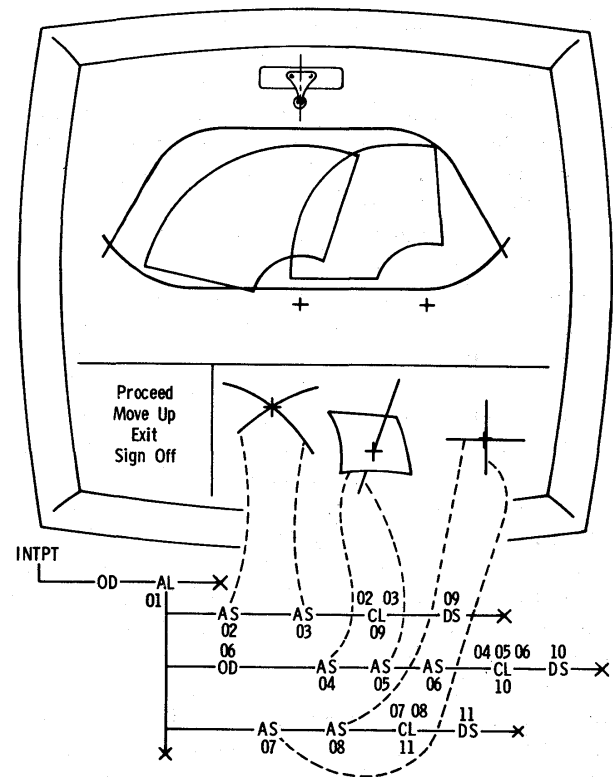


Figure 3—Example operator

this symbol is a data-type used for selective enabling. The number 02 below the AS is called a data code. The data code is no different from a conventional parameter name except it is a more compact way to display the parameter. If the parameter name is desired, it can be displayed along with data declarations and all other information about the symbol. The data code also represents an index into the data table (Figure 5) used at execution time to store the data values assigned to each parameter in the operator. If a complete parameter cross reference is desired that data code may be selected and all references to that parameter will be shown by intensifying all diagram symbols using that parameter. The data code number is a sequence of numbers generated by the system and assigned to symbols requiring a data code as they are added to the diagram by the programmer.

The third symbol, a CL symbol, represents a subroutine call. The subroutine name is stored internally with the symbol and may be displayed if requested by the user. The data code 02 from the first symbol and data code 03 from the second symbol are shown above the CL. This shows 02 and 03 to be input items for the

CL. Below the CL in the first path is data code 09. This is the output data, the intersection point, from the subroutine.

The DS symbol, next on the first alternative path, means that a display of a computed data item is to be made in the work area. Data code 09 above the DS symbol indicates that this parameter is to be displayed.

All symbols in the logic diagram are interconnected by a solid line. To follow this line from symbol to symbol will show a logic flow within the operator. A sequence of interconnected symbols will end with an X symbol representing a path end. The two AS symbols, the CL and DS symbol of the first path are interconnected and terminated with the X signifying a complete path for one choice of the alternative. When the X is reached the logic flow returns to the next higher level and continues. An X symbol on the highest level path terminates the operator. The logic diagram differs from the standard flow chart in that a path of symbols has level significance and the logic flow automatically moves to the next higher level when the path end is reached. An operator logic diagram also has a larger set of symbols than the average flow chart has kinds of polygons.

Further explanation of the alternative (AL) symbol can now be given. It is possible to make displayed items a part of more than one illustration, but regrettably, this is not demonstrated in the example. There must be at least one unique item to each AL path to allow the system to determine which alternative the user intends to use from the selection of a unique combination of items. The unwanted alternatives are removed from the operator picture area, further aiding the user by deleting possible erroneous choices. The remaining pictogram can then be enriched by additional required parameters of lesser importance. This is shown in the example by the OD, operator display, in the middle AL path.

The OD symbol allows the programmer to determine the composition of the operator area. Operators may have one or more operator pictures and additions may be made to the current picture. The OD symbol serves both of these functions. Any OD on the principal or highest level path means that the current picture is to be wiped out and a new composition begins. An OD serving this function is shown in the example immediately after the name INTPT. The second OD in the example on the middle alternative path shows the second function of an OD. This OD is a tolerance value, unimportant to the user of the operator until that path is chosen so the display for that parameter is held back until needed. The name symbol, INTPT in the example, is the initializer for the operator containing required limits and constants.

The Operator Programming Language Compiler

allows the programmer to start building up the logic diagram in the work area symbol by symbol. At any desired time, the logic diagram can be shifted to the operator area and an operator picture built up in the work area. Light pen associations between diagram symbols and items on display in the work area build up the operator graphic interface. There are many sources for displays that may become parts of operator pictures including a picture library, digitizing physical models, parts of work area displays resulting from a design session and a sketchpad facility.

The operator programmer can make rapid and frequent changes to his operators. The display for the operator picture can be altered in any way and any change desired can be made to the logic diagrams. The language compiler will automatically update the internal structure to reflect the changes.

The operators can be executed with a minimum of the logic diagram programmed so it can be built "cut and try." Our DAC experience shows that this is much faster for this type of work than a conventional procedural language that requires careful preplanning of all logic and all too often complete rewriting to incorporate modifications.

## PROCEDURE EXECUTION

### *Interpretive execution*

A module of the operating system called the Executor proceeds from symbol to symbol performing the execution as in conventional interpreters. Major symbols like NAME, OD, AL, AS and CL each have subroutines to perform all the functions required. In this example, the NAME processor sets up tables that drive the operating system. These are illustrated in Figure 5 and discussed briefly in the operating system section.

After initialization the OD module is called. This clears the operator area display and searches the operator structure for symbols with displays, e.g., an AS symbol. The display information is built up until another OD symbol is found or the lowest path end comes up. The picture is then painted on the screen. In the example the operator picture would be painted as shown in Figure 2.

### *Suggested input sequence*

In our example the next symbol is the AL and the suggested input sequence scheme comes into play. This is a prompting mechanism to aid the user with an unfamiliar operator and also to reduce the number of

light pen selections that must be made. Its real function is to make the first selection of each association for the user. The first path of the AL and the first AS (with data code 02 below) is suggested. Externally, the line in the pictogram corresponding to that AS is intensified, selective enabling is set up based on the data type stored with the AS symbol, and the system waits for the user's work area light pen selection to complete the association. However, all other parameters in the work area remain enabled and the user may override the suggestion. The suggested input sequence is then interrupted by selecting the parameter he requires. If this changes AL paths the selected path is used for suggestion, otherwise the system reverts to where the sequence was interrupted. In the windshield design example above, where we suggested the operator was used to intersect the rear view mirror center line with the windshield surface, the first selection of the line in the middle pictogram of the operator picture would be an interruption to the suggested input sequence. The sequence would be shifted from the upper path of the AL to the middle path. The suggested input sequence is determined entirely by programming, according to the order of symbols in the logic structure. The CL, call, symbol causes the Executor to check whether all parameters required for the call have been given values so that the subroutine may be called.

The CL symbol has a feature not available for standard subroutine calls. It may be designated as "once only" or "continuous" execution. The "once only" form is the standard subroutine call but "continuous" means that as long as any one input parameter is continually being given a new data value the subroutine will continue to be reexecuted allowing a dynamic change to an affected part of the design in the work picture. This allows the user to make many small adjustments to his work picture and monitor the result.

#### *Operator internal structure*

A portion of the internal APL structure and corresponding logic diagram is shown in Figure 4. The language compiler, as directed by the logic diagram built by the programmer, generates the internal structure by executing APL<sup>9</sup> statements.

The structure is composed of blocks of information or entities and sets of entities with at least one kind of entity for each symbol in the language. The entities can be connected together into a set by internal pointers to the next entity and to the previous entity in the set. Entities may be members of one or more sets. There is an entity type corresponding to each symbol type in the operator logic diagrams.

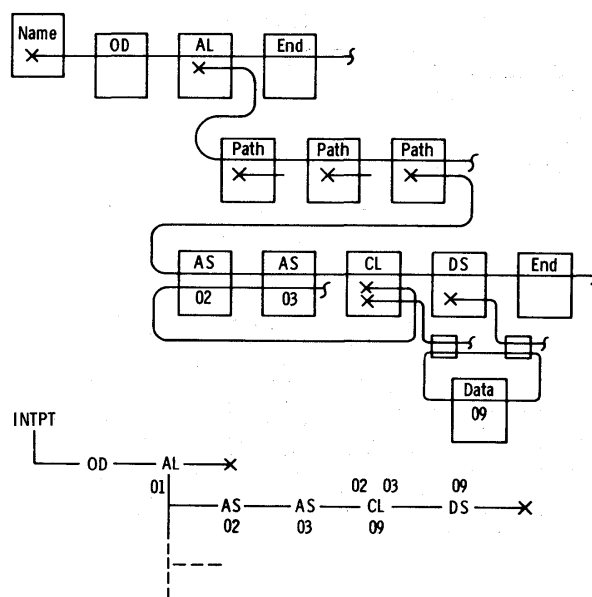


Figure 4—Internal APL structure

There are many ways an operator structure may be modified. The entities can be changed, deleted or hooked up in sets in various ways. These are local structure changes with the advantage that they do not require regeneration of the rest of the structure.

In the example of Figure 4, the logic diagram symbols are each translated to an entity type in the internal structure while logic flow and data flow determine the set connections. Connected to the AL entity are its three alternative paths and most of the entities for the intersection of skew lines path are shown. Also shown in Figure 4 are two node blocks which serve to connect the DATA entity (data code 09) to a set in the CL entity and another set, but with the same set name, in the DS entity.<sup>9</sup>

#### *Hierarchical program structures*

A very important symbol not shown in the example allows an existing operator to become a substructure to a higher level new structure being programmed. It is the primitive operator or PO symbol. An existing structure, when included in another structure, is a lower hierarchical structure and therefore primitive relative to the higher level structure. The PO is like an external subroutine with procedural language programs. This means any operator structure can become a substructure to another operator's structure. Large complicated

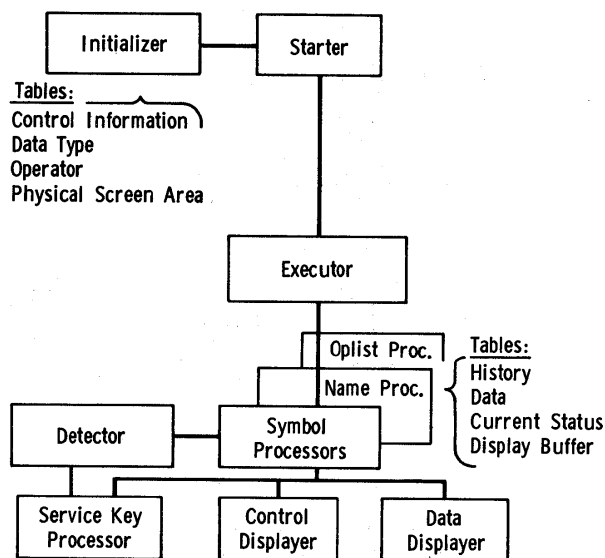


Figure 5—Console operating system modules

structures can be built up in modular fashion from small simple structures.

#### *Automatic retrospective programming*

As a problem is being solved using operator after operator, the operating system can be told to keep a history trace, i.e., to "capture" the procedure. The system builds a simple structure and generates a new PO symbol in that structure for each new operator as it is called. The system tables generated for execution<sup>10</sup> of that operator are saved with the PO. When this simple structure is complete, just a string of PO symbols, it is modifiable in all the standard ways for operator logic diagram generation and modification. Thus a new operator is generated without programming a logic structure, symbol by symbol, in the standard way. The new operator as a simple linking of existing structures was automatically generated by the system from the history trace that was kept. It is retrospective in showing all steps to the new procedure including all data used as input and even the false starts that were abandoned.

### CONSOLE OPERATING SYSTEM

#### *System modules and driver tables*

The interpretive processing of the operator internal structures is done by system modules driven by two

groups of tables as shown in Figure 5. One group of tables contains information concerning a particular application, screen formatting, data types, graphic orders for control area display, function button assignments, etc. This information remains constant during execution and thus these tables augmenting all the individual operator structures may be shared between a number of consoles working on the same application. The second group contains information about the individual console user and the current operator being executed. The use of a standard set of system modules which are controlled by tables allows an application to be tailored to its particular needs. At the same time a uniform approach to the environment is enforced. People working in a particular application are never "surprised" by the kinds of reactions that are expected from them in using their operators just because some operator programmer wished for notoriety by giving his operators a unique set of conventions.

The initializer allocates and initializes the tables universal to an application. This initialization information is supplied by a programmer using one of the programming operators. Different initializing information is called in if a switch is made from one application to another. The control information table determines the control area display. The data-type table indicates types of data to be permitted when setting up for selective enabling for an association. The physical screen area table formats the screen into work area, operator area and control area. The operator table builds up a list of operators used when the user requests automatic retrospective programming.

The name processor starts interpretation of a particular operator. The tables it keeps are for the current operator only. They are saved if a history is being kept, but otherwise freed when the operator is completed.

The executor module carries the structure interpreting along from symbol to symbol. It determines from the entity encountered what processor to call. If a user steps out of the suggested input sequence, the executor must make the change in logic flow and then, when the association is completed, revert to the suggested input sequence.

### PROJECT DEVELOPMENT STATUS AND PLANS

An experimental implementation of the key system modules and major symbol processors has been made for the IBM 360/67, a paged time-shared machine, with several 2250 III graphic consoles attached. Positive results were shown for the proposed internal structure

for operators and system management of the three functional areas of the screen.

Further development for the operator programming language is in progress. The power of the language needs to be increased to include dynamic expressions for the console user while in a problem-solving mode. These are basic arithmetic expressions on numerical data. A higher level expression permitting application defined expressions on the application dependent variables is also under consideration. An example of the higher level expression is the automatic calling up by the system of the two skew line intersection point operator when a point item is needed as data and the console user has just selected two skew lines. More development for the automatic retrospective programming facility is also planned.

## SUMMARY

The treatment of design problems at a graphic console demands communication between man and machine in a manner which is natural to the user so that he may concentrate on finding application problem solutions. This type of interaction can be achieved by presenting the user with pictograms symbolizing concepts and allowing him to select data by making an association between elements of the pictogram and items of the work picture.

Through selective enabling and reactive displays the number of erroneous selections can be greatly reduced. Since data selections may take place in any order, the operators which manipulate the data differ from normal programs in that the statements are not executed in a predetermined order.

The programming of these operators is done in the same graphic environment as application problem solving using the same operator techniques. Through a recording technique the sequence of operations at a console may be captured and used as new operators.

The operating system will manage three functional areas of the screen and permit extensive user interaction with the system to control problem execution and the screen displays. A uniformity of environment can be achieved throughout all the programs of an application that will make the user more at ease and minimize learning to use new programs.

## ACKNOWLEDGMENTS

The techniques and ideas presented in this paper have been developed over a period of time at General Motors Research Laboratories as a cooperative effort of many people working on DAC I and successor projects. I thank Michael Marcotty for his assistance in the preparation of the material for this paper.

## REFERENCES

- 1 B W BOEM et al  
*POGO: Programmer-Oriented Graphics Operation*  
Proc of the Fall Joint Computer Conference Vol 34 pp 321-330 AFIPS Press Montvale New Jersey 1969
- 2 W M NEWMAN  
*A high-level programming system for a remote time-shared graphics terminal*  
Conference on Computer Graphics University of Illinois Urbana Illinois April 1969
- 3 A VAN DAM  
*Hypertext editing system for the /360*  
Conference on Computer Graphics University of Illinois Urbana Illinois April 1969
- 4 E L JACKS  
*A laboratory for the study of graphical man-machine communication*  
Proc of the Fall Joint Computer Conference Vol 26 pp 343-350 Spartan Books Baltimore Maryland 1964
- 5 B HARGREAVES et al  
*Image processing hardware for a man-machine graphical communication system*  
Proc of the Fall Joint Computer Conference Vol 26 pp 363-386 Spartan Books Baltimore Maryland 1964
- 6 P A KOLERS  
*Some formal characteristics of pictograms*  
American Scientist Vol 57 No 3 pp 348-363 1969
- 7 J D JOYCE M J CIANCIOLO  
*Reactive displays: improving man-machine communication*  
Proc of the Fall Joint Computer Conference Vol 31 pp 713-721 Thompson Book Company Washington D C 1967
- 8 G G DODD  
*APL—A language for associative data handling in PL/I*  
Proc of the Fall Joint Computer Conference Vol 29 pp 677-684 Spartan Books Washington D C 1966
- 9 G G DODD  
*APL—Associative programming language user's manual*  
Research Publication GMR 622 General Motors Corporation Warren Michigan
- 10 G G DODD  
*Associative information techniques*  
Associative Aspects of Problem Solving p 51  
American Elsevier Publishing Company Inc  
New York New York 1970

